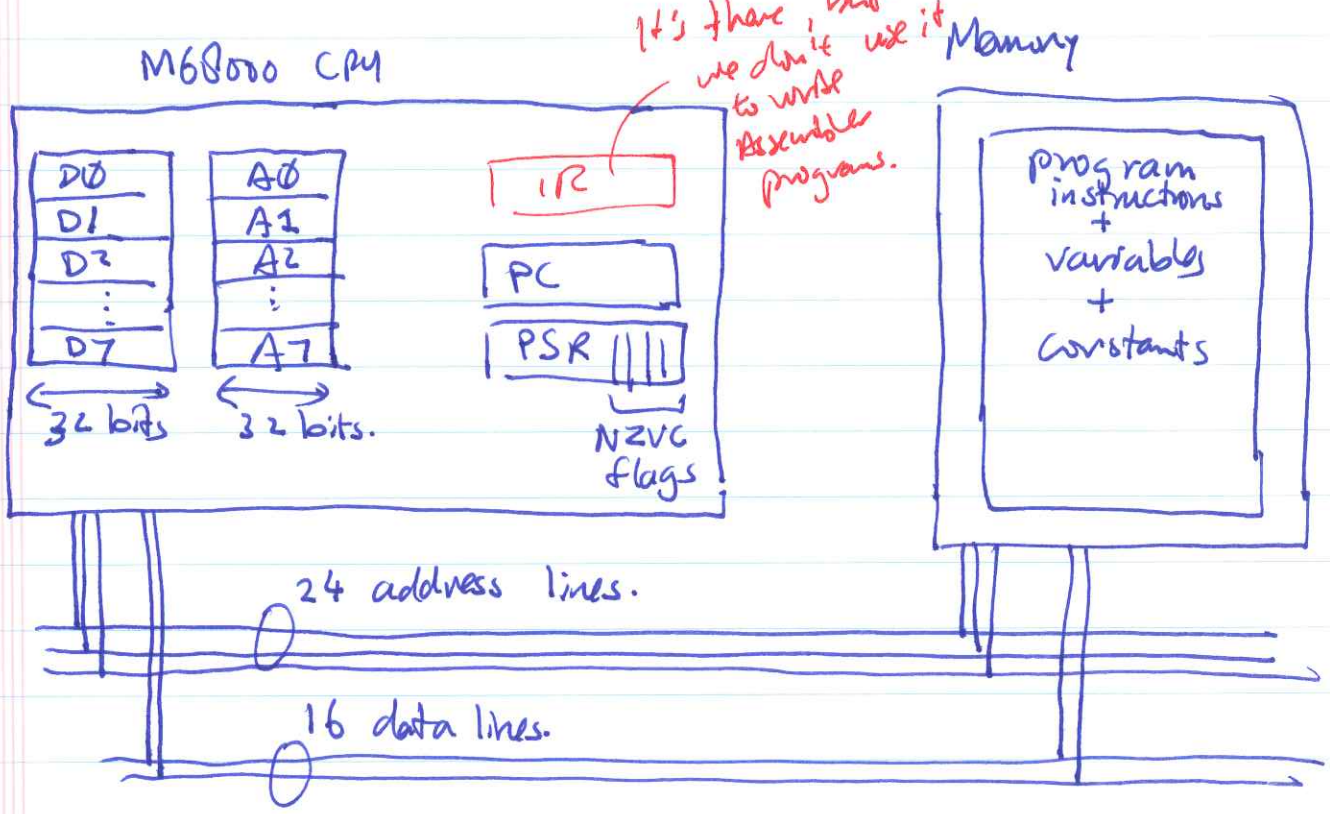
(1) [5]

## Programming the M68000

Programmer's view of the M68000 CPU:




M68000 Registers:

- 8 Data registers named D0, D1, ... D7
- 8 Address reg's named A0, A1, ... A7.

A7 is special. It's the "stack pointer" (more later).

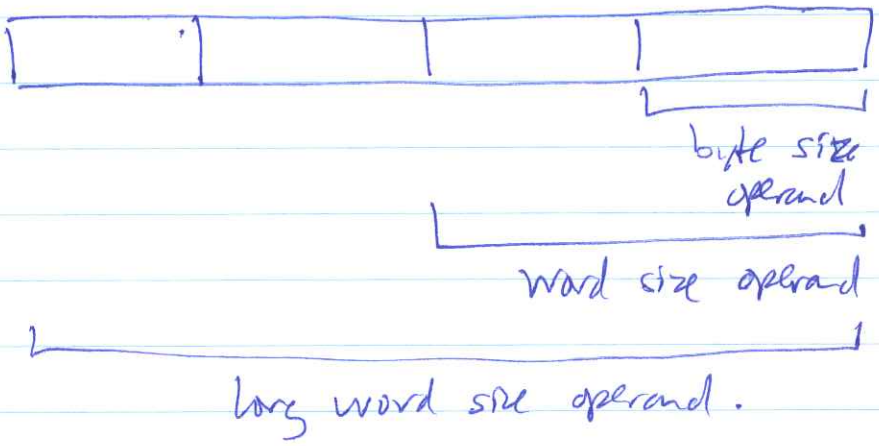
- each register is 32 bits long.

## Data registers

- Used in arithmetic & logic operations.
- can hold/used as: byte size
  word size
  or long word size operands

location of operands:

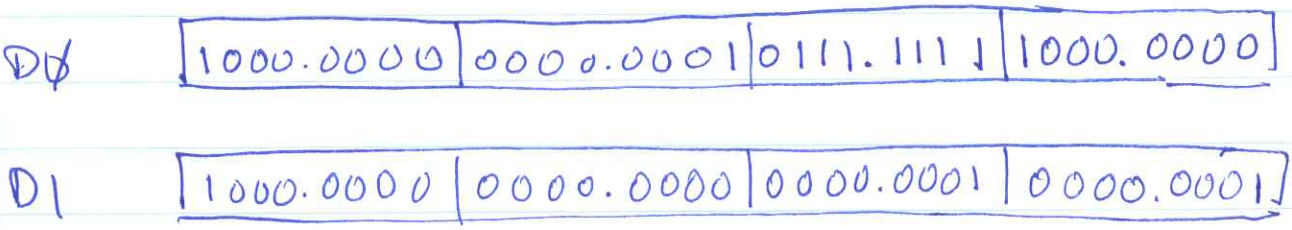
Register:




- byte in M68000 = 8 bits
  word in M68000 = 16 bits
  long word in M68000 = 32 bits.

- Operations that change the content of a data register will set the N, Z, V, C flag according to the outcome of the operation,

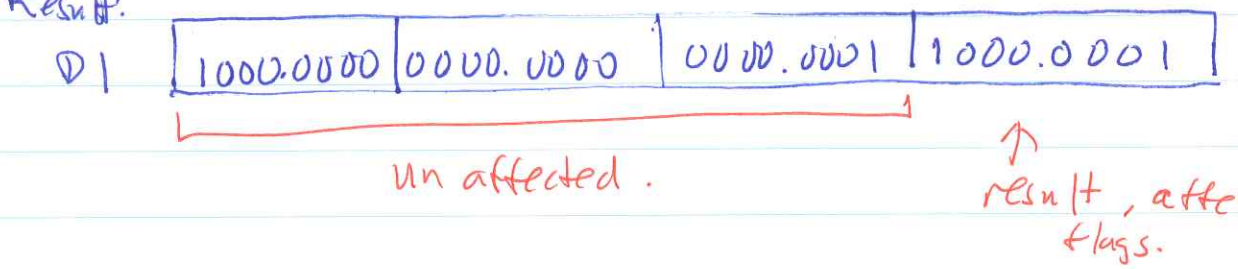
Example:




Instruction:

ADD.B D0, D1 "add byte size operands in D0 & D1, & put sum in D1"

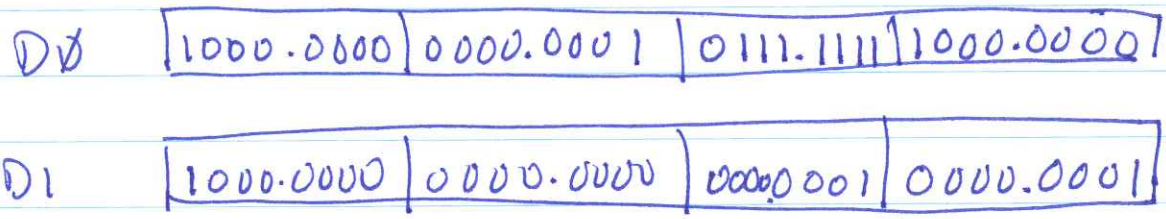
Result.



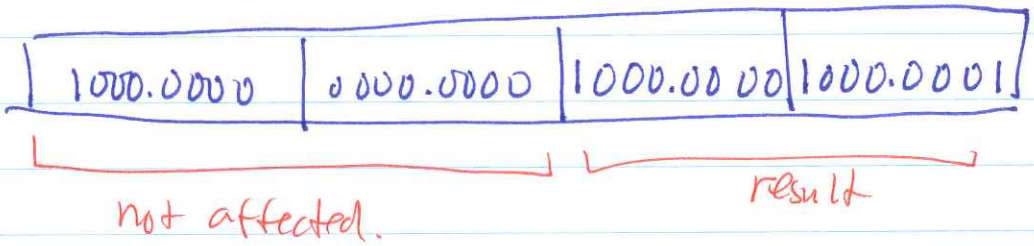

flags: N = 1 (negative result)
Z = 0
V = 0
C = 0.

Instruction:

ADD.W D0, D1 "add word size operands in D0 & D1, put sum in D1"!




Result:




flags: N = 1
Z = 0
V = 1 ← overflow: pos + pos → neg!
C = 0

NEW.

Demos:

- A-reg-operands.s
- reg-byte-op.s
- reg-word-op.s
- ~~reg-longword-op.s~~

Use Byung's GUI!!!

Instruction:

ADD.L D0, D1 "add long word size operands in D0 & D1 and put sum in D1

Before instruction

| | | | | |
|---|---|---|---|---|
| D0: | 1000.0000 | 0000.0001 | 0111.1111 | 1000.0000 |
| D1: | 1000.0000 | 0000.0000 | 0000.0001 | 0000.0001 |

Result: (after instr)

| | | | | |
|---|---|---|---|---|
| D1: | 0000.0000 | 0000.0000 | 1000.0000 | 1000.0001 |

operand.

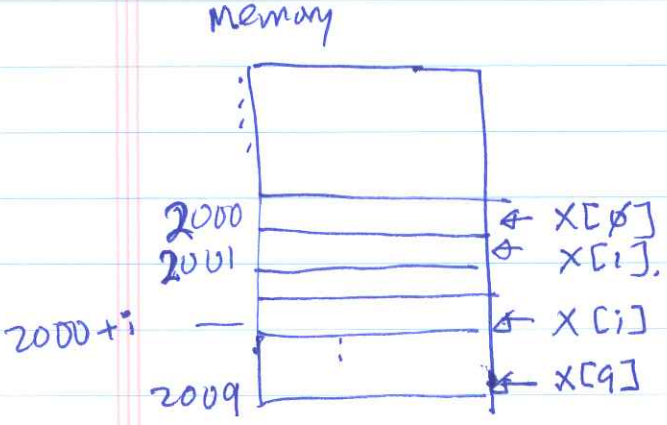
N = 0
Z = 0
V = 1
C = 1

## Address registers

- 8 address registers named A0, A1, ... A7
- Never use A7, it is reserved to used as M68000's system stack pointer (later).
- Primary purpose of Addr. registers is to hold memory addresses
- Complex data structures are accessed through the starting location of the data structure in memory.
  The starting location is stored in an address register.

Example: char x[10];

The array elements are stored consecutively in memory; starting at some location (address), say 2000.



When we write

$$X[0] = 4;$$

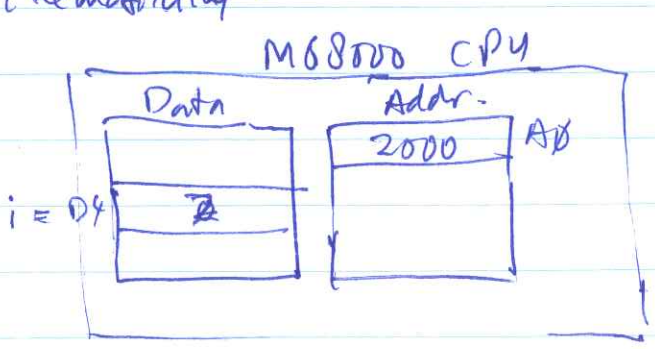
in C, the computer will put the binary number $(0000.0100)_2$ in the memory location 2000.

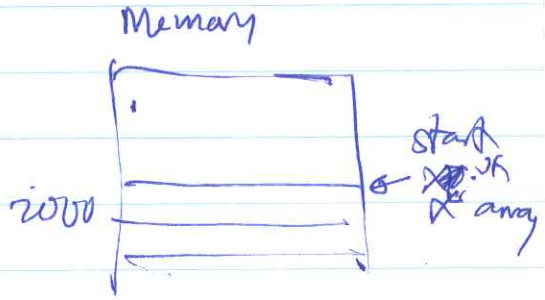
We see that variable X[i] is located at memory address

$$\underset{\text{starting point (or base)}}{2000} + \underset{\text{offset.}}{i}$$

Address registers hold the starting point (fixed). We add the offset i to the address reg. to get to an array variable.

Schematically:



- The address register A0, A1 ..., A6 are general purpose (not dedicated for any purpose). (not reserved)

But addr. reg. A7 is special. (reserved)
A7 is the stack Pointer, designed to implement procedure call & return (save location for the return address (old PC value)).
More on the stack later.

- All operations on address registers will (always) affect the whole registers (affect all 32 bits of the registers). - only long word size operands !!!
(uses)

If word size data (16 bits) are used as operand in an operation involving an address register, the value of the word size data is SIGN EXTENDED to 32 bits before the operation is performed.

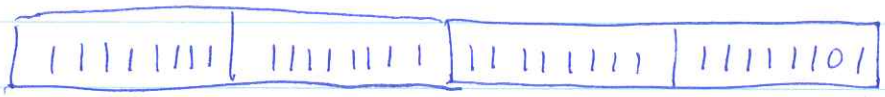
What do you mean?

eg: A0 | 00000000 | 00000000 | 00000000 | 00000100 | 4

D0 | 00000000 | 00000000 | 11111111 | 11111101 | -3.

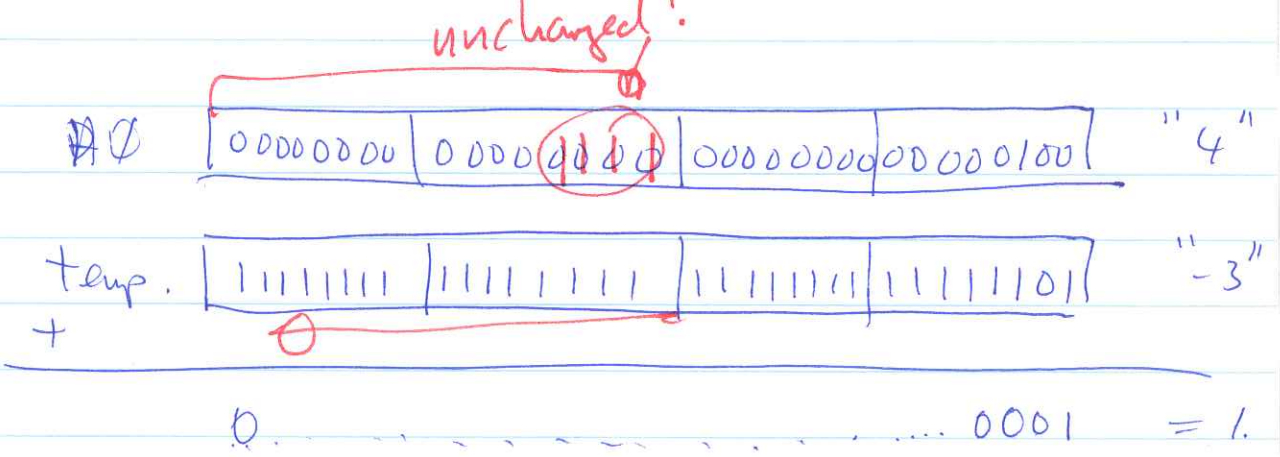
ADD D0, A0 = Add word size operand in D0 to A0.

Since the operand in D0 is word size, we must that it's value (16 bits) & sign extend it:

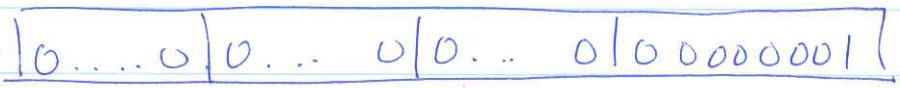
temp: 


Notice that the left most 16 bits in D0 is ignored! in a word size operation!

Then the temp value is added to A0 using full 32 bit operation!



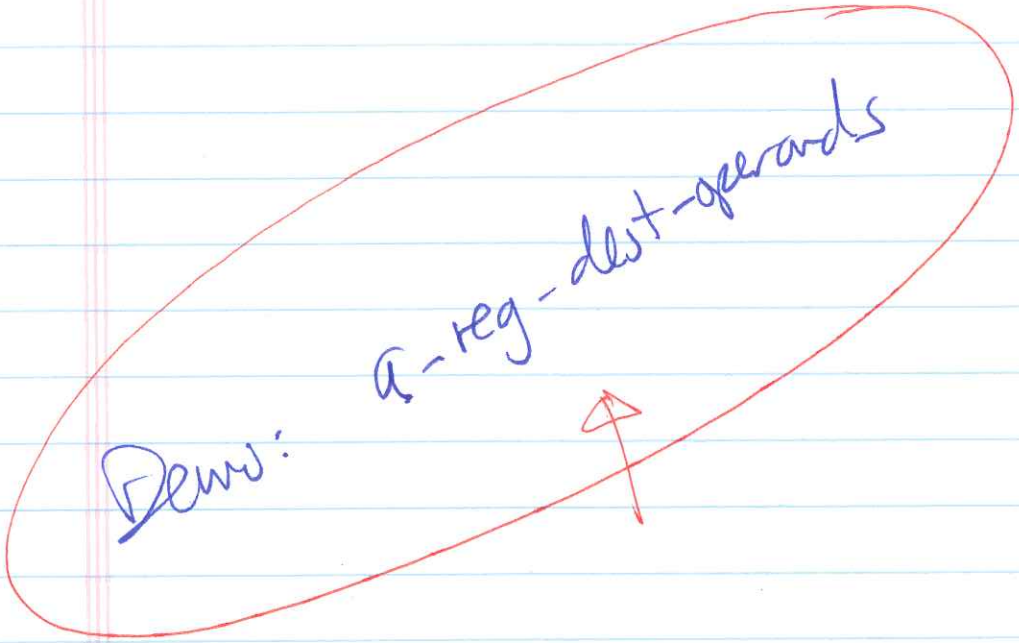

Result:

A0 = 

- Note: operations on address registers do not affect the status of the MC68000's condition code flags.

- Note: byte size operands are not allowed in operations involving an address registers.

(And recall: word size operands are sign-extended before the operation is performed).



## Operands in memory:

- Memory contains variables & constants which are used:

  (1) source operand in computer instructions.

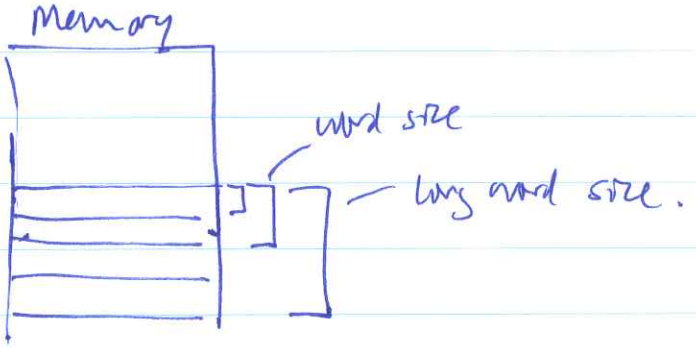
  (2) storage locations for results.

- Byte size operands/results are stored in 1 memory location.

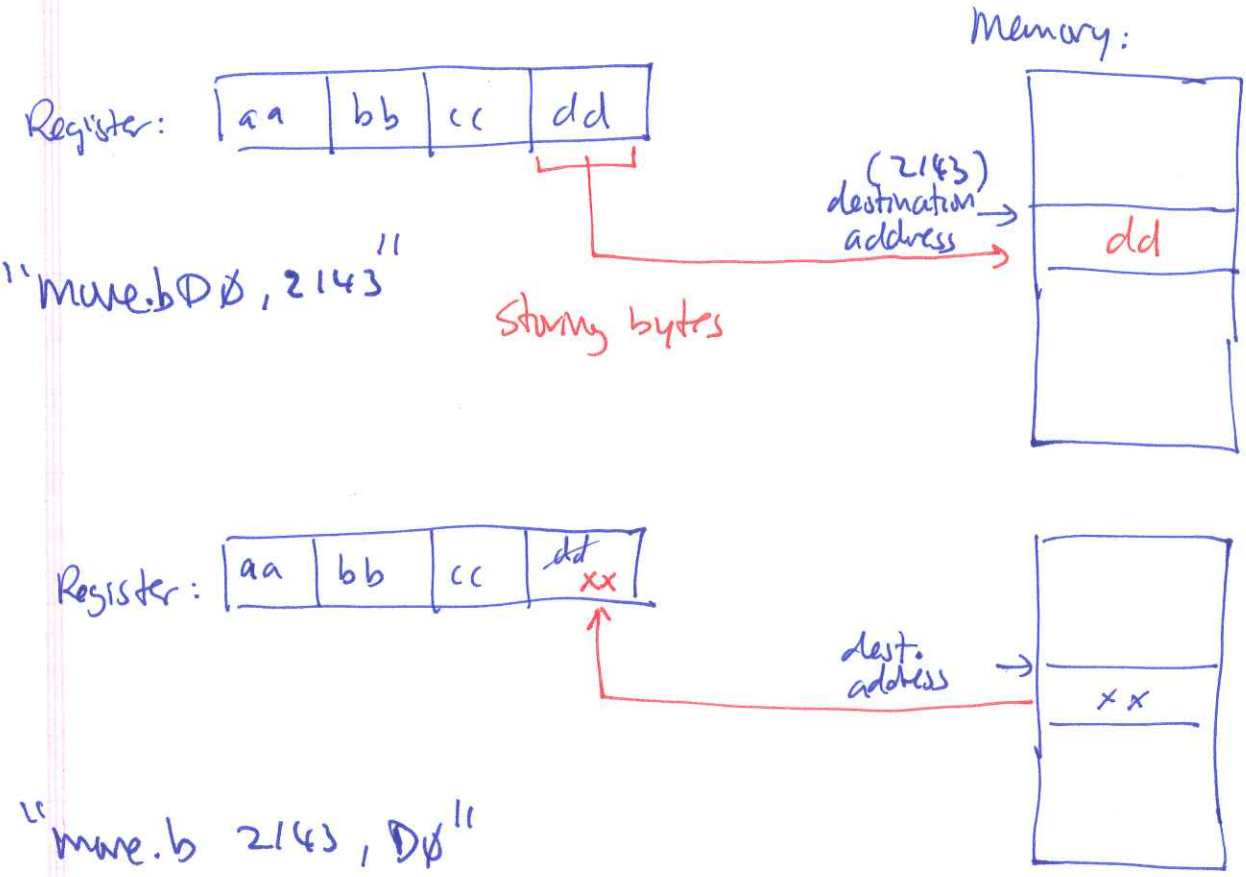
  Word size operands/results are stored in 2 consecutive memory locations, (start must be even)

  long word size operands/results are stored in 4 consecutive memory locations (start must be even).

  Schematically:



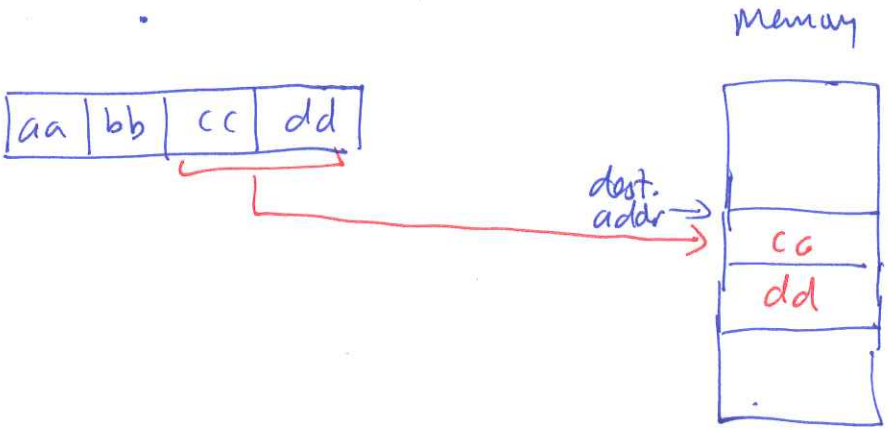
## Storing & retrieving bytes:



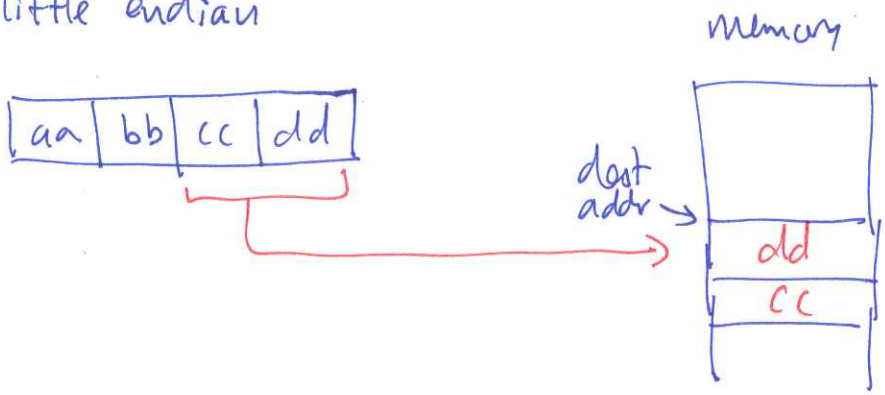
## Storing & retrieving words

- Two ways to store words in consecutive format in general:

(1) big endian: (Gulliver's Travels)




(2) little endian



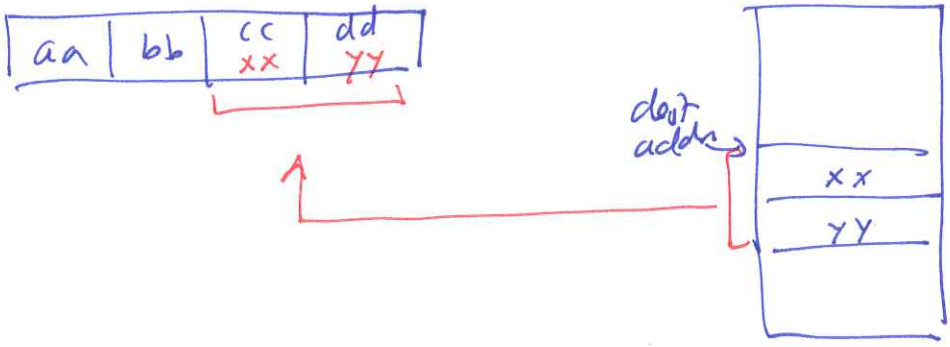

- M68000 & other Motorola CPU's use big endian (SPARC also)

- Intel CPUs (80x86) uses little endian.

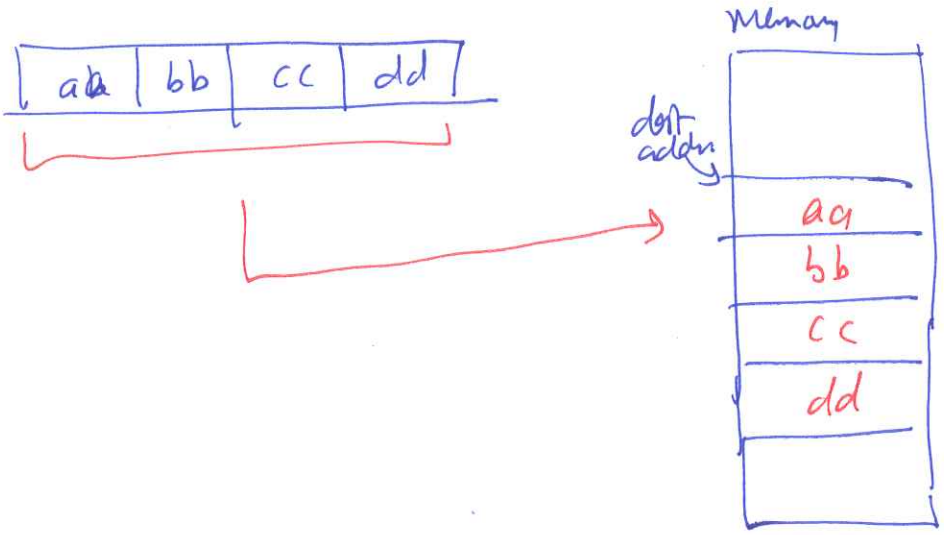
Note: destination addr. for word storage & retrieval in memory MUST be an even address

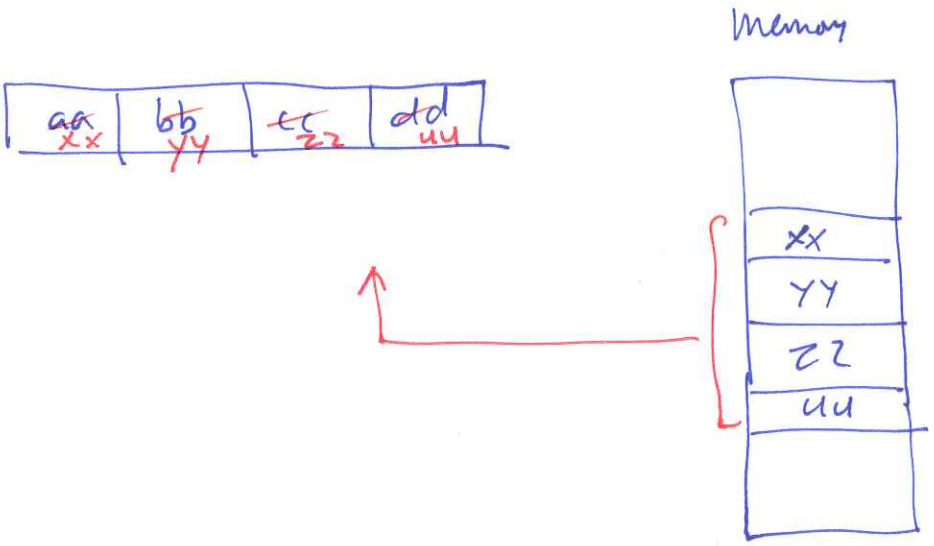
(System requirement).

- Retrieving words from memory (Big Endian).



- Retrieval & storage of long words (M68000)




Demo

mem-operand.S

watch memory location at: 161E: AA BB CC DD

## Programming caveat:

- Always store / retrieve the right (depends on your chosen size) size of operand!

eg: wrong is:

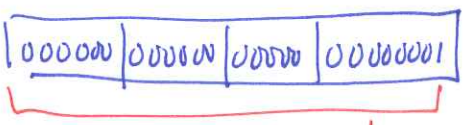
store byte size
retrieve word size

store word size
retrieve byte size

moral:

if you store something of size X
you must (better) retrieve that thing
of size X.

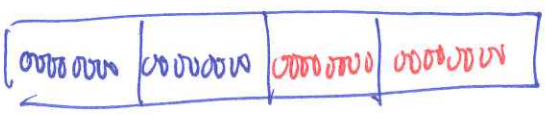
eg:



"g"

store long:

later: retrieve word



"Ø"

Oops...

# Machine level programming

Recall: a machine instruction is a binary pattern that encodes that instruction.

Computers can only execute machine instructions.

Programs written in a high level progr. lang. (like C) must be first converted to machine instructions before it can be executed by the computer.

- Programming in machine instruction is hard because the patterns of 1's and 0's that constitutes machine instructions are meaningless to humans.

To make life a bit easier on the programmers, people have develop mnemonic codes to represent machine instructions.

eg: if | 11110000 | 1010 0101 | encodes the instruction that ADDs D0 to D1 & put result into D1

the corresponding mnemonic code would be:

ADD D0, D1 to represent that instruction.

- The mnemonic codes are called ASSEMBLER codes. or ASSEMBLER instruction

  There is exactly one assembler code for one machine instruction.

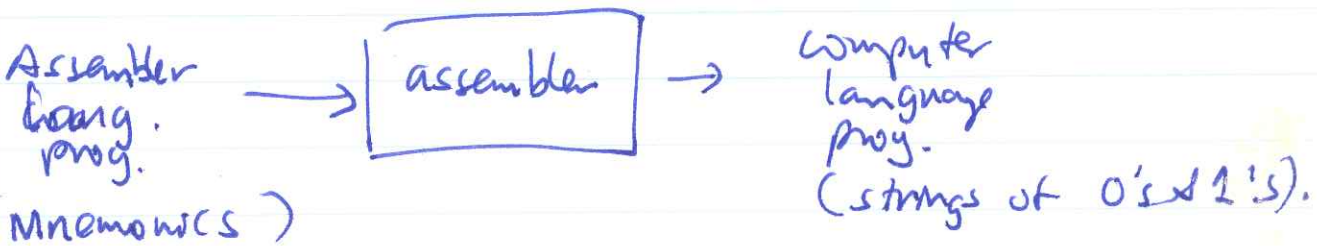
  ~~Assembler code are made~~

  Correspondingly: each assembler instruction is translated into exactly 1 machine instruction.

- Programming in assembler language means to write programs in assembler code.

  Assembler language programs are converted into machine language programs before it can be executed (very much like high level progr. lang. (eg. C) programs).

  The translator program for assembler language programs is called an assembler



## Programming the M68000

- Unlike programming in a high level programming language where the programmer can write his/her program WITHOUT the knowledge of the computer that runs the program, programming in assembler requires the knowledge of the computer that execute the program.

  Reason: you are infact writing machine language so you must know: ~~what kind of registers the cs~~

  (1) What kind of instructions are available.

  (2) How many & what kind of registers are available.

- We will ~~program~~ learn to program two CPU's:

  Motorola 68000 M68000

  ~~SPARC~~

  SUN's SPARC

SKIP

- Why program in assembler codes ?

  (1) programmer has access to all the features of the computer. (Assembler code is heaven for hackers)

  (2) programming device drivers (programs that interact with input & Output (I/O) devices) must be (partly) done in assembler code.

  (3) programs written in assembler are usually smaller & faster than programs written in a high level language which is not optimized.

  (Crucial is time-constraint applications like monitoring a boiler's temperature).

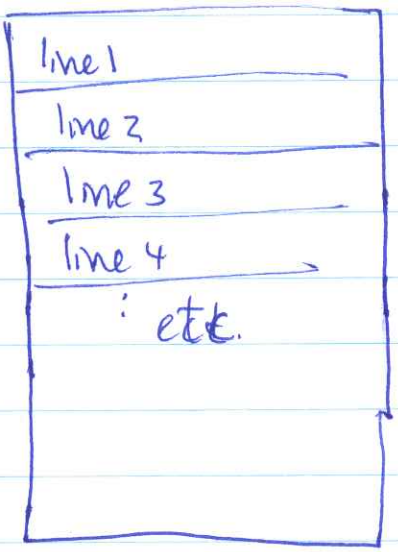
- Drawback of assembler language programming:

  (1) Errors are easily made.

  (2) Hard to read & debug.

# Assembler programming

- An assembler program looks like this:




- Each line of the assembler program contains one of the following:

  (1) an assembler directive

  (2) an assembler instruction (assembler code)

  (3) a comment.

("prime directive")

# Assembler Directives:

- Assembler directives are "hints" or "instructions" given to the assembler.

  Assembler directives do not produce machine instructions.

- Assembler directives are used to:

  (1) Reserve storage space for variables.

  (2) Define symbolic constants.

  (3) Tell assembler to produce instruction to be loaded in a specific location in memory.

  (4) Terminate assembly process.

- ORG : instruct assembler to produce code for a specific location.

Syntax: ORG <value>.

example: ORG 1000 – instruct assembler to produce code that is to be put at location 1000 & onward for execution.

(CS255: starting org 1600H)
↳ automatic!

NOTE: Assembler directives must NOT start @ column 1 of a line. (explained late
column 1 = label !!!)

- END : causes termination of the assembly process. i.e. stop the assembler from translating mnemonic codes to machine ~~code~~ instructions. – used to denote end of assemble program.

Assembler progr. structure:

ORG ....
[ Assemble program
END

- EQU : Used to define symbolic constants.
  i.e. to associate a value to a name.

  Related feature in C: #define MAX 4
  Pascal: const MAX = 4;

Syntax:

<label> EQU <value>

- associate <value> to the name <label>

eg: MAX EQU 4

<label> starts with letter, followed by zero or more letters and/or digits.
(Some assemblers allow '_' underscore also).

<label> can be defined in 2 ways:

(1) Start at column 1
(So any name that start at column 1 is assumed to be a label

(2) Start in any column, terminated by colon (:)

Column 1

eg: MAX EQU 4.

MAX: EQU 4

MAX: EQU 4

MAX EQU 4 → BAD!

## Specifying numeric constants in different bases

Decimal : 17 -18

Binary : %10001
or %00010001

Octal : @21

Hexadecimal : $11

## Specifying ASCII codes

'A' = Ascii code for letter A
= 65

- Note: how to specify constants is assembler dependent. Find out on each system!!!

Story:

Higher level programming languages provide programmer will structured data to help make programming easy.

eg: arrays
structures (records) (struct)
linked lists
etc.

These structures ar logical.

When you program in a low level language like assembler, you only see what's provided by the computer:

You see the memory

The memory consists of an array of bytes.

That's exactly what you will use to "create" logical data structures!

- The M68000 provides 2 assembler directives to define data structures:

  (1) Uninitialized data

  eg. in C: int i;

  The value of the variable is undefined.

  (2) Initialized data.

  eg, in C: int i = 4;

  The value of the variable i is initialized to 4.

- ~~Understand what the~~

- Question:

  What does the declaration:

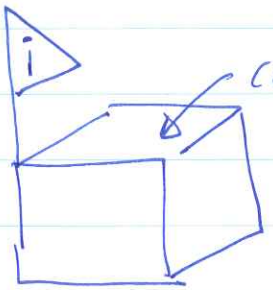
  int i;
  or int i=4;

  do for you in C???

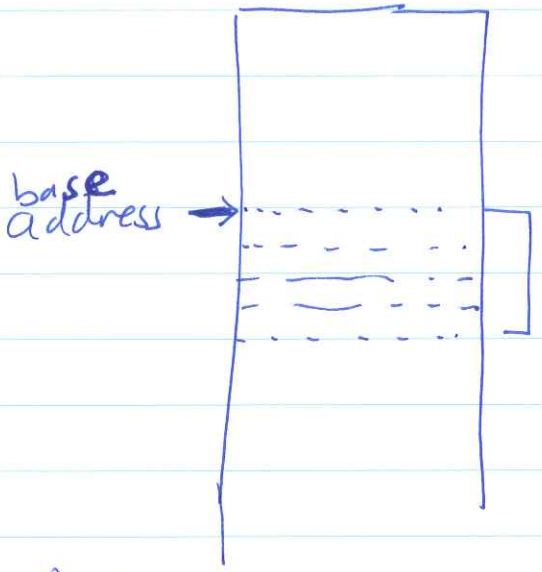
Answer:

(1) it "creates" a box to hold an integer

(2) it "labels" the box with the name (identifier) ;:




- The box in reality is a portion of the memory reserved to store a binary number (integer!).




if variable occupies more than one (consecutive) byte of memory, it is identified by the addr. of the first byte ("base addr")

start

The location of these bytes is important for locating the value of the integer.

- **DS** : Define Storage

demo: ds.s

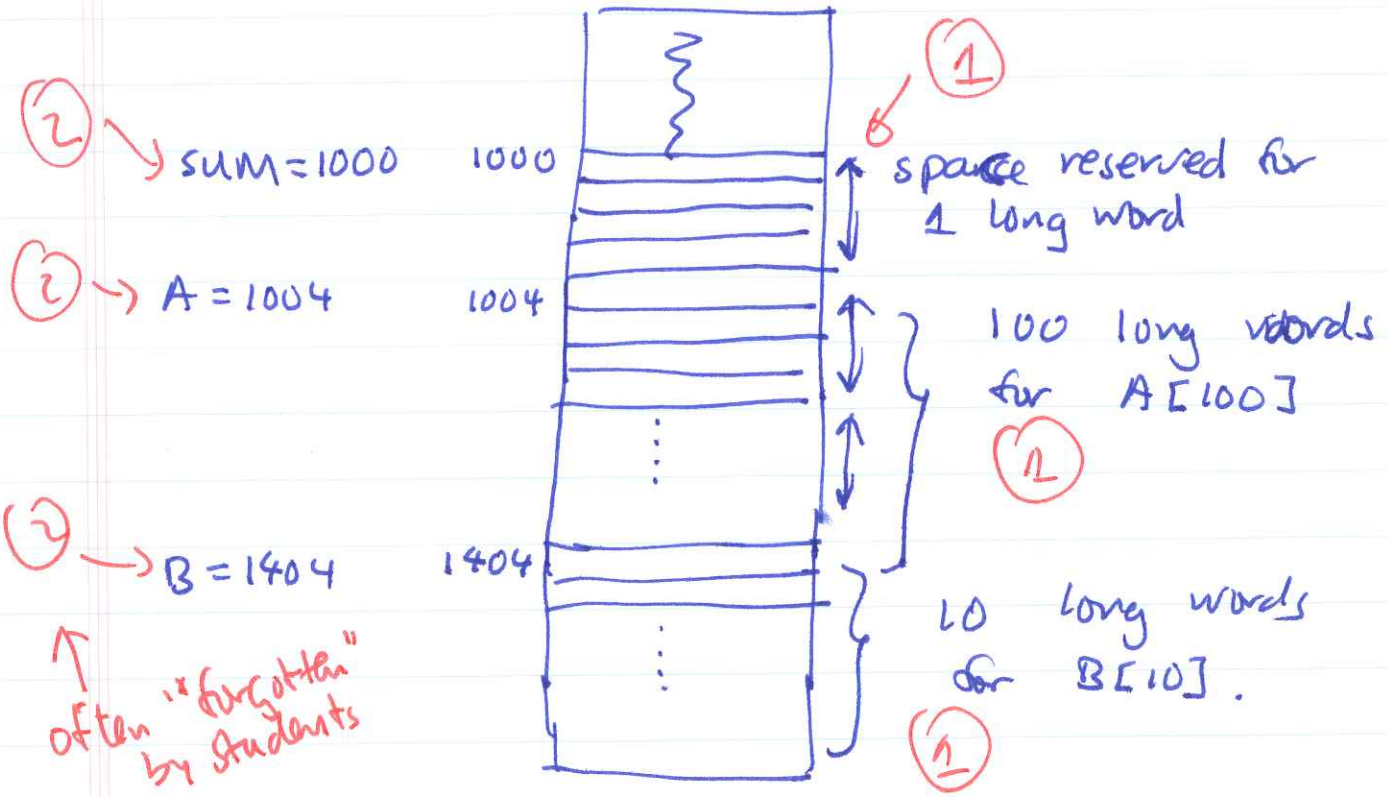
Syntax : <label> DS.<size> n

effect : (1) reserve n memory locations for n variables. (2) assign the starting (base) address to the symbolic name <label>
(address of the first byte)

eg :

```
sum:    DS.L    1
A:      DS.L    100
B:      DS.L    10
```

Result : in memory :




- Where the computer allocates the memory for the variables is irrelevant, as long as we can find it back and not memory occupied for 2 purposes!

demo : dc.s

- **DC** : Define storage and initialize it with some value.

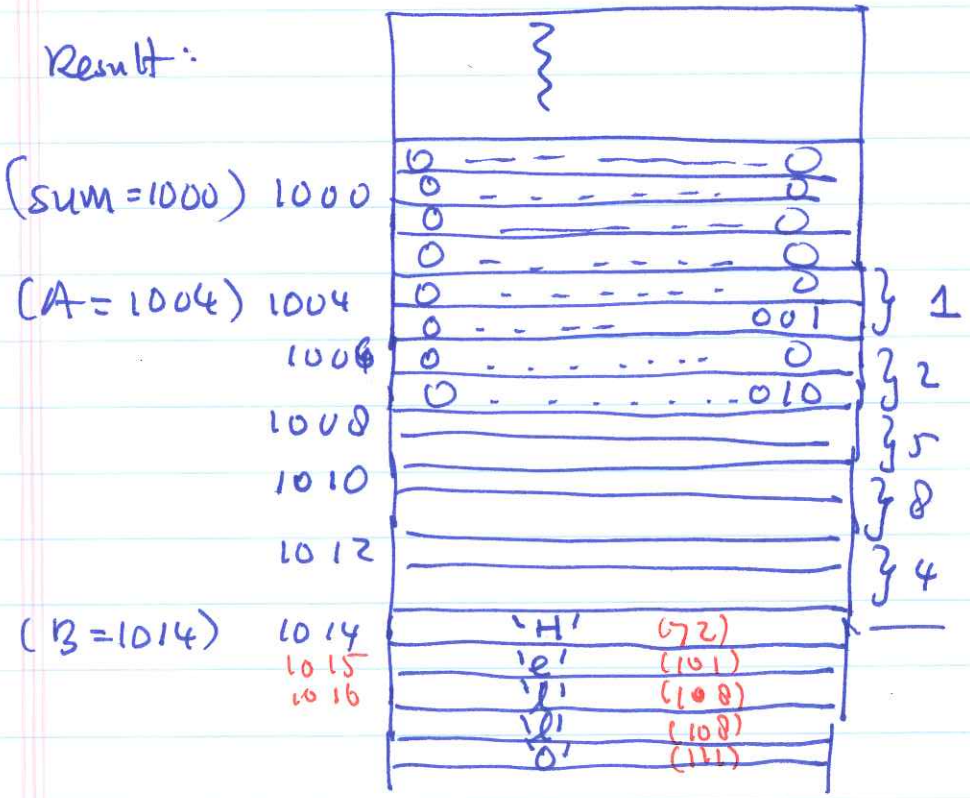
syntax : <label> DC.(size) $n_1, n_2, n_3 \ldots$

effect : reserve memory locations & initiate it with values $n_1, n_2, n_3 \ldots$
The starting address is assigned to the symbolic name <label>

eg :
```
sum :  DC.L  0
A   :  DC.W  1, 2, 5, 8, 4
B   :  DC.B  'Hello'
```

↑ values put in the memory locations !!!

Result :



# How to define variables with DS & DC in C

(1) Simple variables:

(stress: effect of defining variables → reserve memory!)

```
int i,j,k;          i: DS.L 1
                    j: DS.L 1
                    k: DS.L 1

int i=0;            i: DC.L 0

short n;            n: DS.W 1

short n=10;         n: DC.W 10
```

Quiz students →

```
unsigned int x;     x: DS.L 1
```

(no diff. in size, diff. in use!) ← same as signed int!

(2) Arrays

```
int A[10];          A: DS.L 10

int A[]={10,5,6,7}  A: DC.L 10,5,6,7

char str[10];       str: DS.B 10

Char Str[]="Hello"; str: DC.B 'Hello',0
```

'\0' terminated string (in C).

class Person
{ char name[40];
char addr[40];
int salary;
}...

Person like; → like: ds.b 24

- **Structures:**

```
struct { char name[10];
         char addr[10];
         int salary;
       } person;
```

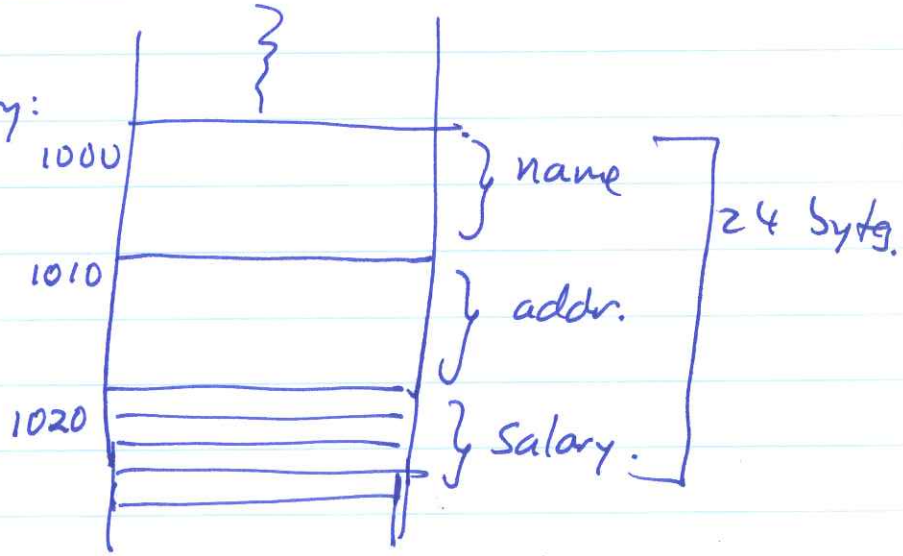
person: DS.B 24

Question: what happened to name, addr, salary?

Answer: field in structures are usually NOT identified by name, but by a relative offset from the start of the structure.

The start of the structure is identified by name.

So: in memory: (person = 1000)

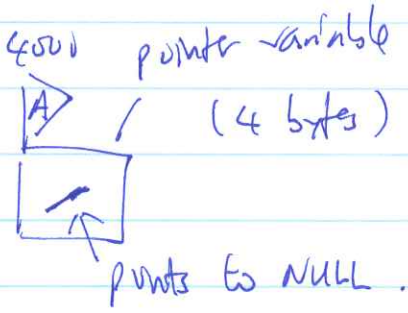



Array of structures:

```
struct { char name[10];
         char addr[10];
         int sal
       } person[10];
```

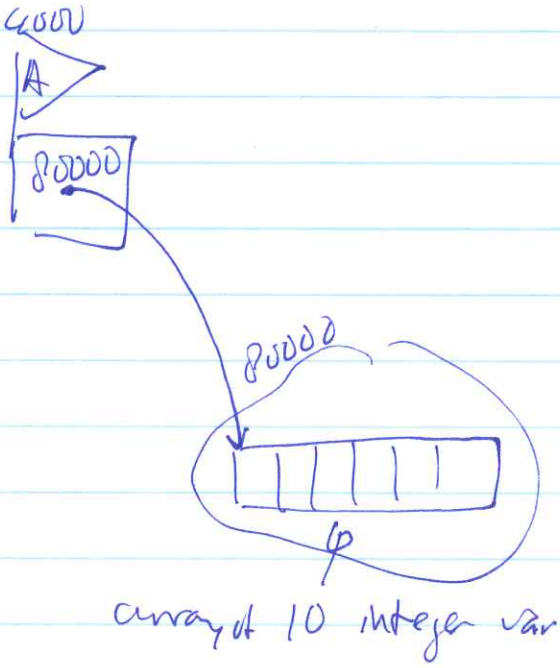
→ person: DS.B 240

## Arrays in Java:




`int [] A;` ⇒

points to NULL.




~~int~~ `A = new int [10];` ⇒

- Arrays that we create in assembler looks like this: